WL-TR-95-1167

SYNTHESIZING FIELD PROGRAMMABLE GATE ARRAY CIRCUITRY USING C++

JOHN T. SPILLANE
DR MICHAEL A. ZMUDA

AUGUST 1995

FINAL REPORT FOR 03/31/95-08/31/95

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7409

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

JOHN T. SPILLANE, Design Engineer
Data & Signal Processing Section

JERRY L. COVERT, Chief
Information Processing
Technology Branch

CHARLES H. KRUEGER, Chief
Systems Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAAT, WPAFB, OH 45433- 7409 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	AUG 1995	FINAL 03/31/95--08/31/95

4. TITLE AND SUBTITLE SYNTHESIZING FIELD PROGRAMMABLE GATE ARRAY CIRCUITRY USING C++

5. FUNDING NUMBERS
C
PE 61102
PR 2300
TA AA
WU 05

6. AUTHOR(S) JOHN T. SPILLANE
DR MICHAEL A. ZMUDA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7409

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7409

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-95-1167

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In recent years, systems which use Field Programmable Gate Arrays (FPGAs) to perform computations have been shown to match or even exceed super computer levels of performance. These FPGA based custom computing machines (FCCMs) take advantage of an FPGA's gate-level reconfigurability to implement instructions and architectures specific to the problem being solved. The ability to tailor the hardware to a specific problem gives FCCMs a great speed advantage over general purpose processors with fixed instruction sets. Currently, one of the largest obstacles to the use of an FCCM is program or instruction set development.

Since an FCCM is programmed at the gate level, the programmer must have detailed knowledge of both the algorithm to be implemented and how to implement necessary operations (e.g. addition, subtraction, and multiplication) required by the algorithm in the FCCM's hardware. This design process is comparable to programming in assembly language, though hardware instruction design is arguably more difficult. Recent research efforts have investigated the creation of a novel, symbiotic compiler which can simplify the development of programs for FCCMs by hiding the hardware instruction set generation from the programmer in much the same way a traditional compiler isolates the user from a specific machine's assembly language. This paper presents the details of the internal operation of this FCCM compiler and plans for future work.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Field Programmable Gate Array, Computer Architecture, Synchronous Data Flow	19
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Contents

1 Introduction 1
1.1 Computational Model 1
1.2 FPGAs as Co-Processors 2
2 Symbiotic C++ Compiler 5
2.1 The `Var` Class 6
2.1.1 Example of `Var` Usage 7
2.2 FCCM Board Level Classes 8
2.3 FPGAs and Derived Classes 9
2.3.1 Pins and Package 9
2.3.2 CLB Array Sizing 10
2.3.3 Basic Functions of the `xc4k` Class 10
2.3.4 Linking `Vars` to FPGA objects 10
3 Future Work 12
3.1 Additional Features 12
3.1.1 Mixed Vendor Boards 12
3.2 Optimizations 13
3.2.1 Optimization Based on a TmpVar class 13
3.3 Placement Based On Data Flow 13
4 Conclusions 14

Figures

1. Simple Data Flow Graph 1
2. Data Flow Graph reduced to Synchronous Data Flow 2
3. Design Flow for using FPGAs as Co-Processors 2
4. Hamming Distance Calculation 3
5. C Code for Hamming Distance Function 3
6. Circuit Implementation of Hamming Distance Calculation 4
7. Traditional Compiler Flow 5
8. c3 Compilation Flow 5
9. Sample Operation Flow 6
10. Example c3 Program 7
11. FPGA Class Hierarchy 9
12. c3 Code Fragment 11
13. FPGA Class Definitions with Virtual Base Class `General_FPGA` 12

1.0 Introduction

Field Programmable Gate Arrays (FPGAs) are hardware reconfigurable devices introduced in the mid 1980s as single chip replacements for discrete digital logic components[2]. These devices have become popular by reducing design cycle times, reducing board real-estate required for random logic, and allowing quick changes in the logic after board fabrication. In the late 1980s, engineers began using FPGAs to implement co-processor instructions. During that time the available gate capacities were relatively low when compared with FPGA processor boards which now reach capacities in excess of 300,000 equivalent gates. By the year 2000, FPGA manufacturers predict board capacities of 5 million or more equivalent gates. As FCCMs increase in complexity, the difficulty of creating programs that effectively utilizes system resources also increases. This drives the need for automated programming methods.

1.1 Computational Model

The computational model a system uses to solve problems or compute answers influences decisions made during system design and programming. For example, a program written for a sequential system tends to be different in syntax and/or organization when compared with the same algorithm coded for a parallel machine. The computational model used during the creation of the c3[†] compiler is synchronous data flow (SDF)[4]. In a pure data flow machine [5], results are computed as soon as the inputs to a operation become valid. SDF simplifies the general data flow model by stating that the timing, i.e. when inputs become valid, is known. Unlike data flow, SDF does not reduce latency, but allows high throughput while keeping the problem of compilation more tractable.

Consider the data flow graph shown in Figure 1. In a data flow environment the addition is per-

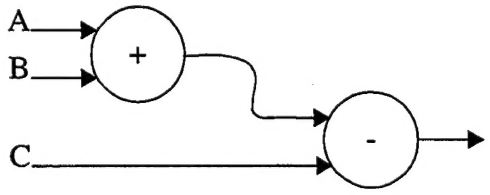

Figure 1: Simple Data Flow Graph

formed as soon as the A and B inputs are available. Likewise, the subtraction is performed as soon as C and the result of the addition operation are available. In SDF, clocked operations are used to simplify synthesis. The SDF graph is shown in Figure 2. This graph assumes that A, B, and C are all initially valid during the same clock cycle. Delay registers are used to ensure proper data alignment in time, e.g. at the input to the subtraction operation. The latency introduced into the operation by these added registers is acceptable only because without them the problem of

---

[†]. The c3 (see-three) compiler is short for CHAMP cc. The Configurable Hardware Algorithm Mappable Preprocessor (CHAMP) is an FCCM based on Xilinx 4000 series technology. For more information of CHAMP see [3]

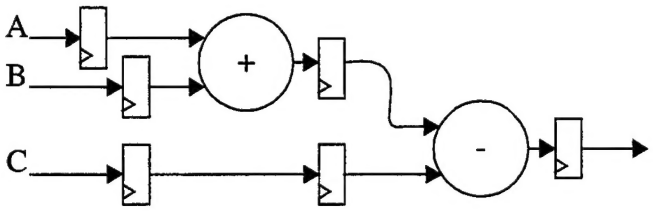

Figure 2: Data Flow Graph reduced to Synchronous Data Flow

compilation and circuit synthesis is greater. Again, since the target system is a FCCM, it is possible to change the computational model at some point in the future.

1.2 FPGAs as Co-Processors

In recent years there have been a number of systems that have effectively used FPGAs as co-processors to reduce program run-time up to 50 fold [6]. The idea for this type of system stems from

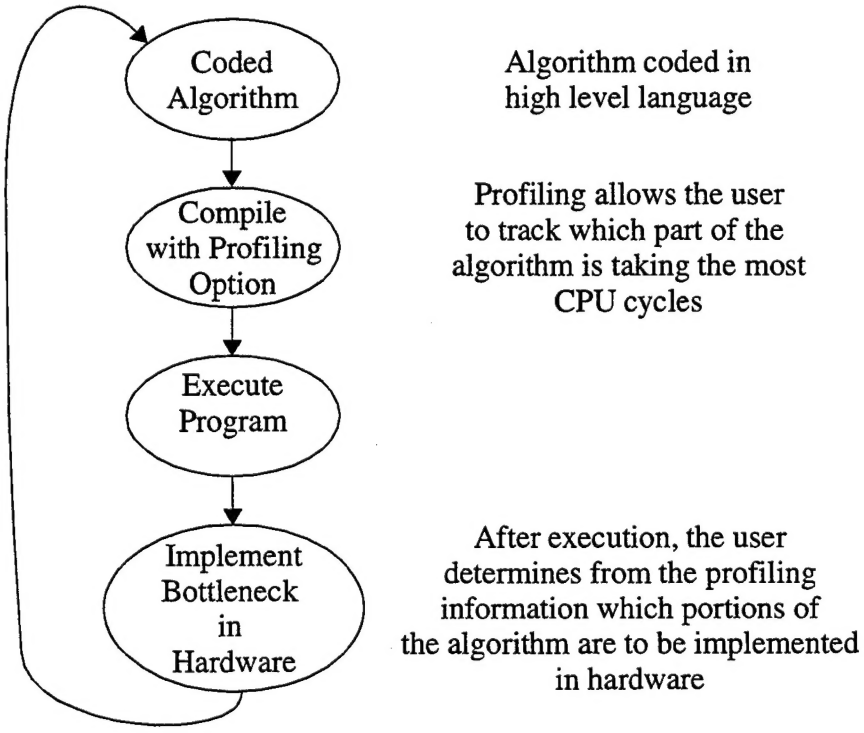

Figure 3: Design Flow for using FPGAs as Co-Processors

what computer scientists have known for quite some time: many programs spend a large percentage of time doing one particular operation or function. A hardware implementation of this speed bottleneck in a co-processor can significantly reduce the total run time of a program. Figure 3 shows the process of locating and implementing speed bottlenecks in hardware. As an example both a software and an FPGA hardware implementation of a Hamming distance function are

examined. Hamming distance is defined as the number of bits that are different in two words of some known, equal length. To calculate the Hamming distance, the two words are compared bit by bit for inequality and the number of ones, i.e. differing bits, in the resulting word are counted. Assuming a word length of eight bits, an example Hamming distance calculation is shown in Figure 4.

```
01001100
11110000
----------
10111100 Hamming Distance = 5
```

Figure 4: Hamming Distance Calculation

A C function to calculate Hamming distance is shown in Figure 5. This function takes two eight

```
unsigned char Hamming_Distance( char a, char b)
{
  char xor_result;
  unsigned char distance = 0;
  int i;

  xor_result = a ^ b;
  for (i=0; i < 8; i++)
  {
    distance += xor_result & 1;
    xor_result <<= 1;
  }
  return(distance);
}
```

Figure 5: C Code for Hamming Distance Function

bit arguments and computes the Hamming distance between the two words. Notice that the return type is also an eight bit value. Although the return value will only take on values in the range [0,8], most compilers do not allow the programmer to choose a type that is only one nibble, four bits, wide. FCCMs are flexible in that a variable may be any number of bits, assuming that the bit width is supported by the internal FPGA architecture. Unfortunately, not only is this low-level design detail available to the FCCM programmer it also plays an important role in creating programs that fit within the confines of a particular FCCM. Elaborating, the traditional programmer often chooses a variable of type 'int', typically represented with sixteen bits, whose value ranges from -32768 to 32767. This type is often chosen even if the range of the object being stored is known to range from [0,10]. It is simply a matter of convenience for the programmer to use type 'int'. In an FCCM, resources are limited and bit widths must often be chosen on an operation by operation basis. This adds a level of complexity to FCCM programming that will not be discussed here.

A circuit implementing the Hamming distance calculation is shown in Figure 6. Again we note

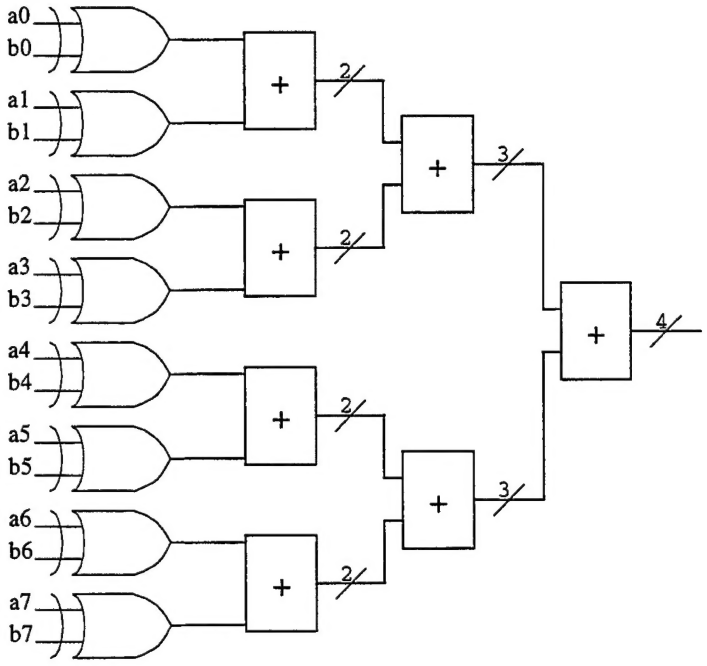

Figure 6: Circuit Implementation of Hamming Distance Calculation

that the FCCM is capable of outputting the result in the minimum number of bits needed for the problem at hand. Assuming that each level of logic is clocked, this circuit would have a latency of four clock cycles plus two clock cycles due to registers at the input and output pins. This results in a total latency of six clock cycles. Examining only the inner loop of the C function that implements the same operation, twenty four operations are needed to solve this problem on a sequential processor. Assuming the FPGA circuit is run at 25Mhz, the traditional processor would need to run at 100Mhz to match the latency of the FPGA implementation. Also, as the bit width of the Hamming Distance arguments increases, circuit growth follows the $\log_2$ of the number of bits in the argument whereas the number of operations required by the sequential processor grows linearly. Thus if the bit width of the arguments were doubled to sixteen, the FPGA circuit would have a latency of five plus two, or seven clock cycles. The sequential loop will have an approximate latency of sixteen times three, or forty-eight clock cycles. Now the sequential processor would need to be approximately seven times faster that the FPGA to give the same latency. This means that the processor is running at 175 Mhz compared with the same 25Mhz FPGA.

Also, since the FPGA circuit is fully pipelined it can accept new inputs each clock cycle and, after the latency period has passed, produce outputs on each clock cycle. So if the task were to compute the Hamming distance on five pairs of sixteen bit numbers, the FPGA circuit would require seven clock cycles as latency until the first result appears and four more clocks to produce the rest of the results for a total of eleven clocks. The sequential machine would require five times forty-eight operations, 240 operations, where the FPGA circuit requires only 11 clock cycles. To compute the five results in a length of time on the order of the 25 Mhz FPGA, the processor would have to run at over 500Mhz. The advantage of the FPGA as compared with the conventional processor is clear.

2.0 Symbiotic C++ Compiler†

To reduce the time, effort, and expertise required to program an FCCM, a symbiotic compiler that creates SDF circuitry from C++ code has been developed. This work stemmed from earlier work in which the process of designing control chips for CHAMP was automated. The symbiotic compiler differs from a traditional compiler in that it has no front-end, i.e. pre-processor and parser. The compilation flow for a traditional compiler is shown in Figure 7. With a traditional compiler,

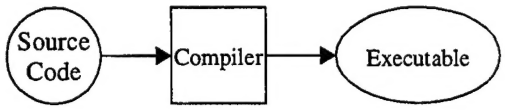

Figure 7: Traditional Compiler Flow

the user inputs source code into the compiler and an executable program is output. The compilation flow for the symbiotic compiler is shown in Figure 8. Here the code for the symbiotic com-

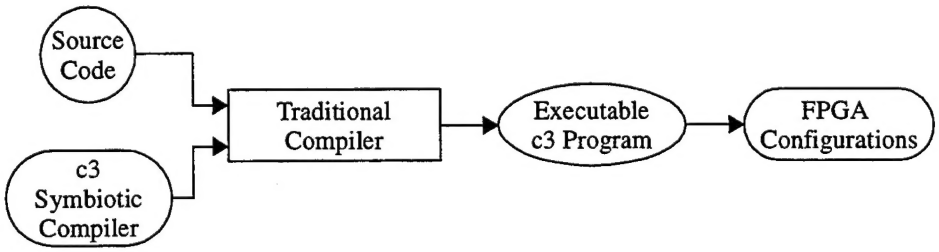

Figure 8: c3 Compilation Flow

piler and the user's source code are compiled together by a traditional compiler. The output of the resultant program is configurations for various FPGAs within the FCCM. The CHAMP cc (c3) compiler is a symbiotic compiler. The main advantage of this type of compiler is ease of development. Since the symbiotic compiler is based on operator overloading, no parsing of input files is required. The drawback to this type of compiler is the limited instruction set that the symbiotic compiler can handle. In C++, only certain operators can be overloaded and no constructs, such as a while loop, are overloadable. Therefore, the c3 compiler can only currently handle logical operations, addition, and subtraction with unsigned integers.

When c3 encounters an operation, the overloaded operation will flow as shown in Figure 9, "Sample Operation Flow". The overloaded operators are found in the `Var` class which is discussed in Section 2.1, "The Var Class", on page 6. The overloaded functions within the `Var` class directly address functions found within the `Target` object which is discussed in Section 2.2, "FCCM Board Level Classes", on page 8. The `Target` object then calls a particular FPGA object in which the operational circuitry will be implemented, in the case of Figure 9 this circuitry would be an

† The following discussion assumes a detailed knowledge of C++. For more information on the C++ language and Object-Oriented Programming see [7][8][9]

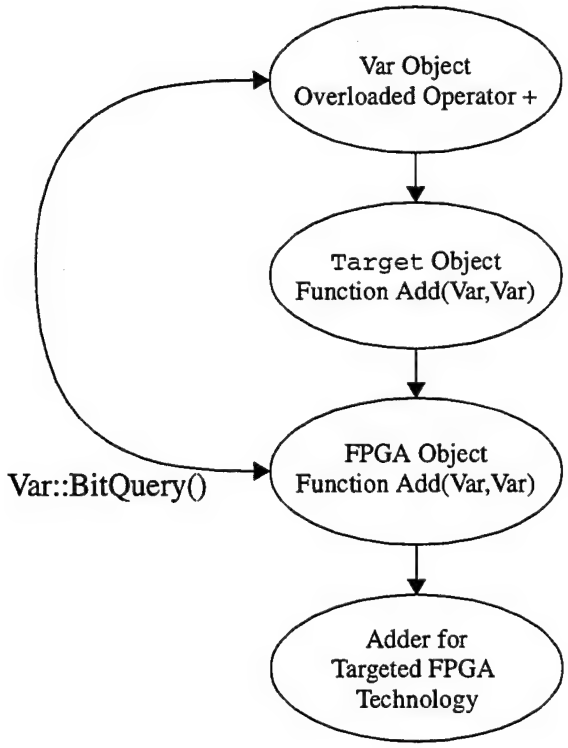

Figure 9: Sample Operation Flow

adder. These FPGA objects and the use of the function `Var::BitQuery()` are discussed in Section 2.3, "FPGAs and Derived Classes", on page 9.

2.1 The `Var` Class

The `Var` class is a data abstraction that allows the user to deal with instructions that operate on variables rather than FCCM gate level instructions operating on single bits. Currently the operators shown in Table 1 are implemented for unsigned integer arithmetic. With continued work, it

Operation	Symbol
Left Shift	<<
Right Shift	>>
Logical AND	&
Logical OR	|
Logical XOR	^
Logical NOT	~

Table 1: Currently Implemented Operations

Operation	Symbol
Addition	+
Subtraction	-

Table 1: Currently Implemented Operations

would be possible to implement signed integer as well as fixed point arithmetic. C++ operators which are currently thought to be possible to implement, but are not as yet implemented are listed in Table 2. Though division is listed as possible, it is improbable that c3 will implement this oper-

Operation	Symbol
Pre/Post Increment	++
Pre/Post Decrement	--
Multiplication	*
Division, Modulo	/,%
Comparisons	==,>,<,>=,<=,!, &&,||,!=

Table 2: Possible Future Operations

ation until FPGA devices are large enough to handle operations the size of division in a fraction of the space available in the device.

2.1.1 Example of `Var` Usage

An example c3 program is shown in Figure 10. In this program the target FCCM is the Xilinx

```
extern xc4kDemoBoard Target;

c3_main()
{
  Var x,y,z;

  x = Target.switches(1,2,3,4);
  y = Target.switches(5,6,7,8);

  z = x & y;

  Target.bin_2_left_7seg(z);

  cout << Target;
}
```

Figure 10: Example c3 Program

4000 series demo board. First notice that this program has no `main()` function. Instead, the `main()` of a c3 program is named `c3_main()`. This function is then called from the `main()` that resides within the c3 compiler code. In the first statement in the example program, `Var x,y,z;`, the user declares three variables of the class `Var`. At this point the C++ program that has resulted from symbiotic compilation has simply called the default `Var` class constructor, `Var::Var()`, three times. The next statement, `x=Target.switches(1,2,3,4)`, shows the use of a `Target` object. `Target` objects will be discussed in detail in Section 2.2; for now it suffices to say that this statement links the variable `x` with four DIP switches found on the 4000 series demo board. Similarly, the statement `y=Target.switches(5,6,7,8)` links the variable `y` with four separate dip switches also on the board. The `switches()` function, which is specific to the Xilinx 4000 series demo board, does such bookkeeping tasks as setting the number of bits, and latency values for both the `x` and `y` variables.

The statement `z=x&y` is the first stand-alone C++ statement in the example program. This statement by itself looks as if it could have come from any C++ program and, as a programmer would expect, this statement assigns to the variable `z` the result of the logical ANDing of the variables `x` and `y`. On a traditional processor, these variables might be of type char (8 bits) or type int (usually 16 bits), but with the `Var` class any bit width, which need not be a power of two, is possible.

This statement, `z=x&y`, is also the first statement in the example program that uses an overloaded operator. As shown in the high level diagram of Figure 9, the `x&y` statement will call the overloaded & operator for the `Var` class, `Var Var::operator &(Var)`. This overloaded operator, in turn, calls a function `Target.and()`. Again, notice that the `Var` class is isolated from the particular `Target` object's technology. That is, the `Target` FCCM is not tied to any particular FPGA technology, although Xilinx 4000 series parts are the only devices for which functionality is currently implemented.

The next statement calls another demo board specific function, `bin_2_left_7seg()`. This function takes a four bit `Var` and converts its value to logical levels suitable for driving the left seven segment display on the demo board. It is important to note that `Target` level functions such as `switches()` and `bin_2_left_7seg()` help to isolate the user from the specific architecture of the `Target` system. In this case, it is no longer necessary for the user to know exactly to which pins the switches on the demo board are connected, or that the LEDs of the demo board's seven segment display are active low. This type of architectural information is included in the `Target` system class definition so that a novice user is able to code programs for the system.

2.2 FCCM Board Level Classes

Figure 9 shows an example object of an FCCM board level class. This object is referred to with the c3 reserved variable name `Target`. The `Target` object serves as the communication layer between the `Var` class and the FPGA class. Each overloaded operator in the `Var` class calls a corresponding member function of the `Target` object. This mechanism allows the user to change the hardware that c3 will generate circuitry for without having to modify the `Var` class's operators. `Target` objects can range in complexity from a single FPGA system, such as the Xilinx 4000 series demo board, to a multiple FPGA system such as CHAMP.

In FCCMs such as CHAMP which contains multiple FPGAs, a c3 board level class must have a partitioner, or board level placement routine, available to it. Currently this functionality is not implemented as all test programs have been compiled for a single FPGA system. It is important to note that board level partitioning in an SDF system should be coupled with logic generation since different partitionings of a problem may require different levels of pipeline alignment due to the registering of inputs and outputs at the chip level.

Once the board level object, referred to with the reserved variable name `Target`, has determined within which FPGA a particular operation will be placed, the object calls the corresponding FPGA operation. Given the adder example from Figure 9, the `Target` object would choose which FPGA on the FCCM will implement the add. Then the `Target` object calls the add function within that FPGA. The FPGA level add function currently places the generated circuitry in a snake-like fashion within the FPGA. Internal FPGA placement based on a data flow diagram is discussed in Section 3.0, "Future Work".

2.3 FPGAs and Derived Classes

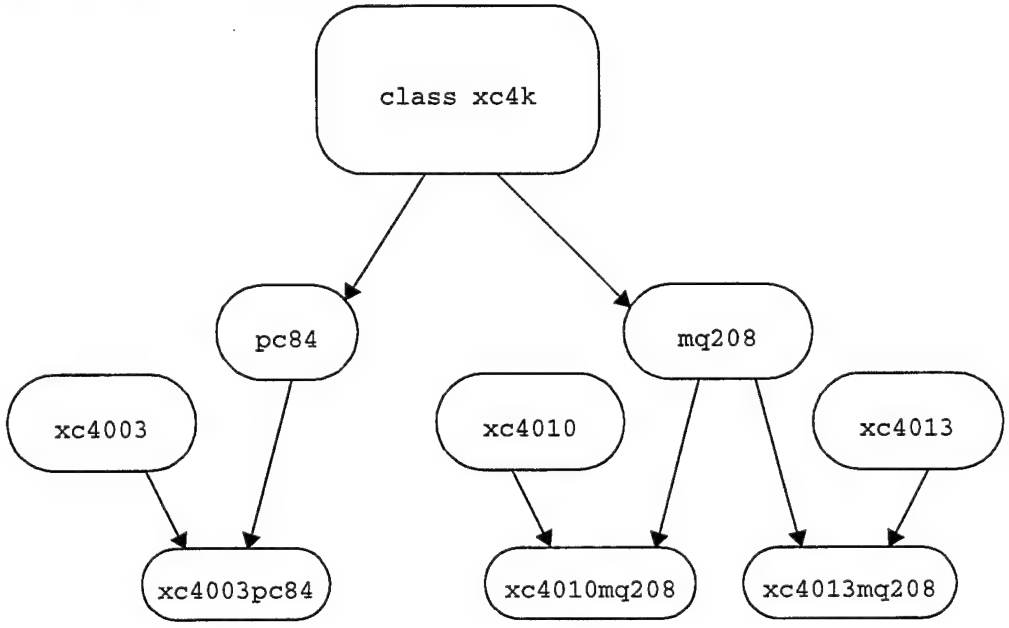

Figure 11: FPGA Class Hierarchy

The class `xc4k` is shown as the top level class in Figure 11. As the top level class, `xc4k` defines how functions are implemented in and the general structure of a Xilinx 4000 series FPGA. The structure, as defined in `xc4k`, consists of a two dimensional array of configurable logic blocks (CLBs) surrounded by what can be reduced to a linear array of input/output blocks (IOBs).

2.3.1 Pins and Package

After the family of FPGAs has been selected, the next level of separation is the number of pins and the type of package for the part. The pins and package class fills the IOB array with informa-

tion specific to the particular combination. For example, the pins that can be used to drive global clock buffers change from part to part. This class is used to track these differences as well as package specific information such as unusable pins.

2.3.2 CLB Array Sizing

The pins and package level class, e.g. `mq208`, is still an abstract class. Objects at this level contain no information regarding the number of CLBs within the part. This parameter, the number of rows and columns within a particular device, forces a design decision. One choice was to have the user declare a part as `mq208 part(24,24)`. Here, the arguments to the mq208 constructor give the number of rows and columns in the device. The second option was to use the C++ class inheritance methodology to place these parameters into their own class from which a device level class would be derived. By doing this the declaration of a Xilinx 4013 part simplifies to `xc4013mq208 part`. It was decided that this method of declaration was more descriptive so the abuse of the class structure, using a class only to hold parameters, is overlooked.

2.3.3 Basic Functions of the `xc4k` Class

There are three functions basic to the `xc4k` class: placement, clock selection, and logic synthesis. The placement function is used to position generated logic inside the FPGA's matrix of CLBs. The clock selection function allows the user to specify which pin, or internal oscillator, will act as the clock for the circuits within the FPGA. The logic synthesis function is actually a set of functions where each function handles the synthesis of a particular type of operation.

2.3.4 Linking `Vars` to FPGA objects

When a particular logic synthesis function within a xc4k object is invoked, the function generally requires information about its operands. If the compiler is executing the statement `a=b+c`, the `xc4k add` function will be called with the `Vars b` and `c` as arguments. The add function will need details such as the bit widths, sign values, and shift values of `b` and `c`. Other functions will require this same type of information about their arguments, though how the bit level information affects the operation implementation can vary.

The function `Var::query_bit(int)` is used to isolate the FPGA class from low-level details of the `Var` class. This function returns a `BitQuery` structure which include information that describes a particular bit of the `Var` as well as on which net that bit of the `Var` is located. Figure 12 shows a c3 code fragment that will be used to investigate the operation of the `Var::query_bit(int)` function. Here b and c are four bit, unsigned integers. The Var b has been shifted left by one bit, and is then added to c. Within the add function there is a loop which will examine bit by bit the two arguments to be added. Code within this loop makes decisions based on the types of bits that are presented to the operation. Listed in Table 3 are the results for

```
Var a,b,c;

b=Target.switches(1,2,3,4);
c=Target.switches(5,6,7,8);

b <<= 1;

a=b+c;
```

Figure 12: c3 Code Fragment

the repeated calls to `Var::query_bit(int)`. Resulting circuitry for the adder is affected by the

Bit Number	`Var b` State	`Var c` State
0	INT_FILL_ZERO	VALID_BIT
1	VALID_BIT	VALID_BIT
2	VALID_BIT	VALID_BIT
3	VALID_BIT	VALID_BIT
4	VALID_BIT	EXTEND_ZERO
5	EXTEND_ZERO	EXTEND_ZERO

Table 3: Example `query_bit()` Results

states of `b` and `c`. For example, the shifting of `b` causes the first bit of that `Var`, bit 0, to have the state INT_FILL_ZERO. In this state, the net returned by `query_bit()` is invalid. The `add` function must know to create a local ground net for use as input to the adder at this stage. Similarly, the adder must understand that when both arguments are in the EXTEND_ZERO state all bits in the `Var` arguments have been examined.

3.0 Future Work

3.1 Additional Features

3.1.1 Mixed Vendor Boards

It is given that there are certain functions which are basic to a particular family of FPGAs and that there are other functions a user may want to implement with FPGAs that may be described optimally by these basic functions. Through the use of classes and derivation, c3 could support a multi-vendor FPGA environment (i.e. a board with FPGAs from more than one vendor). This can be accomplished for some set of instructions by providing core instructions at the vendor family level, while describing more complex functions at a higher class level using virtual definitions for the basic family functions.

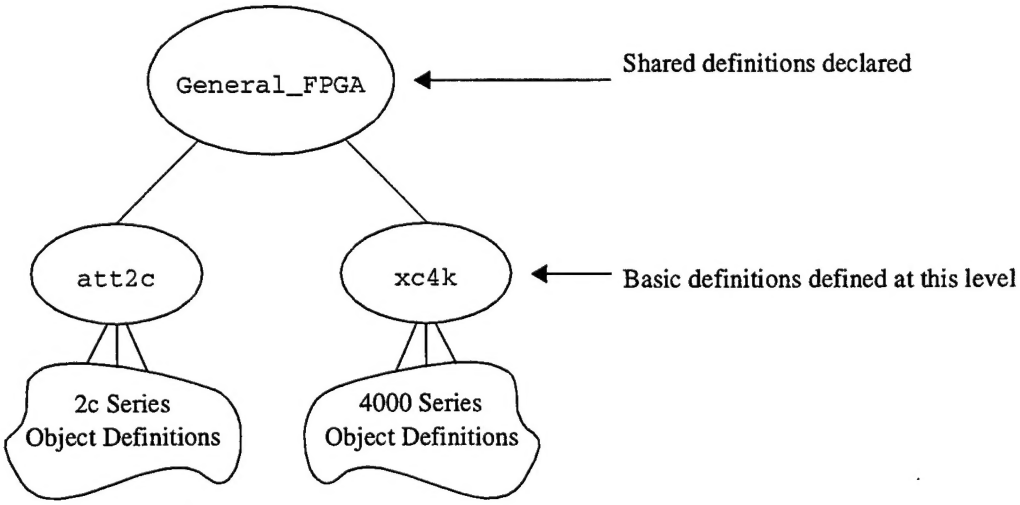

Figure 13: FPGA Class Definitions with Virtual Base Class `General_FPGA`

With a class structure as shown in Figure 13, high level functions such as `HammingDistance()` could be written for the abstract class `General_FPGA`. Once objects derived from this class are fully defined, such as at the xc4010mq208 level, the `HammingDistance()` function would be defined using the basic FPGA functions that were defined at the `xc4k` class level. Similarly, any fully defined vendor part that has the basic operations defined, as shown in Table 1 and Table 2, and is descendant from the `General_FPGA` class will have a defined `HammingDistance()` function.

The `General_FPGA` class concept uses classes to isolate the vendor specific functions, which tend to be very basic functions, from those higher level functions that can be reused across vendor lines. Some functions may not be vendor specific, but cannot be moved to the `General_FPGA` class without optimization at the FPGA level. An example is random logic. Different vendors have different block structures within their FPGAs, e.g. the Xilinx CLB vs. ATT's PFA. Thus

describing a method for a four input AND operation will be different between vendors, or part families which offer course vs. those who offer fine grain building block structures.

3.2 Optimizations

3.2.1 Optimization Based on a TmpVar class

The use of operator overloading in the Var class creates a direct mapping between functional statements in c3 code and circuitry implemented in the FPGA. This direct mapping is not always optimal. Consider the statement a=b&c&d. The circuit that a designer would create for this would be a three input AND gate. In Xilinx 4000 series FPGAs, each bit three input AND gate can be implemented in one half of a CLB. Thus if b,c, and d are each unshifted and four bits wide the three input AND operation can be implemented in only two CLBS. Currently, c3 will implement c&d creating an intermediate result which will then be ANDed with b. This results in four CLBs being used as two input ANDs and two CLBs being used to time align b with the result of c&d for a total of six CLBs. By creating a second class, tmp_var, operations such as this could be optimized for the targeted FPGA architecture. By changing functions such as Var Var::operator &(Var) to TmpVar Var::operator &(Var), and then defining methods TmpVar Var::operator &(TmpVar) and similar overloads for the TmpVar class, c3 could optimize its implementation of logical functions without defining methods such as Var Var::and3(Var,Var).

3.3 Placement Based On Data Flow

Currently the placement function for the xc4k class places generated circuity in a snake-like fashion within the CLB array. This type of placement leads to circuits which may be either unroutable or not meet timing constraints. Having c3 create a run-time data flow diagram and then place based on the origin and destination of signals should create placed designs that are similar to those currently done manually.

4.0 Conclusions

A novel, symbiotic compiler for the creation of FPGA circuitry in the SDF domain from C++ source code has been developed. The compiler is user extensible and includes class structures that can be used as a basis for the development of other FPGA related tools. Unfortunately, the compiler does not completely eliminate the need for knowledge of FPGA based computing. For example, the programmer must understand that the generated circuitry will be fully pipelined and therefore the coded operations will be executed on each clock cycle. Also, if the generated circuitry is not fully functional, debugging may prove difficult. Work to add functionality as discussed in Section 3.0, "Future Work" is ongoing.

[1] D.T. Hoang, "Searching Genetic Databases On Splash 2", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, pp. 185-191, April 1993.

[2] The Programmable Logic Data Book, Xilinx Inc., 1994.

[3] B. Box, "Field Programmable Array Based Reconfigurable Preprocessor", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, pp. 40-48, April 1994.

[4] E.A. Lee, "Synchronous Data Flow", Proceedings of the IEEE, September 1987.

[5] A.H. Veen, "Dataflow Machine Architecture", ACM Computing Surveys, pp. 365-396, December 1986.

[6] P. Athanas, H. Silverman, "Processor reconconfiguration through instruction-set metamorphosis", *IEEE Computer*, pp. 11-18, March 1993.

[7] S.B. Lippman, C++ Primer, Addison-Wesley Publishing Company, 1991.

[8] Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley Publishing Company,1987.

[9] G. Booch, Object Oriented Design With Applications, Benjamin-Cummings Publishing Company, 1991.